Lindenmayer Systems
CS 499 - Assignment 7
Meghan Clark

1 Meghan's Awesome(!!!) L-System Machine

I checked out MASON's L-system demo and decided it was too complex and opaque for the kind of tinkering I wanted to do with L-systems. In particular, I wanted rapid and open-ended options for configuring both the grammar *and* the interpretation. L-systems seemed simple enough, so I implemented my own L-system generator and visualizer in Python. I'm very happy that I did. I ended up with a cool toy - I mean, tool - that does what I want in a way that I (and hopefully others) can easily understand. You can find the code and documentation at https://github.com/mclarkk/L-System.

1.1 Features

Meghan's Awesome(!!!) L-System Machine can generate and interpret strings for basic and bracketed L-systems. You can define an L-system by specifying an alphabet, axiom, and rewrite rules. You can also specify the number of rewrite iterations. The interpretation is done via Python's turtle module. You can control the interpretation by setting the turtle's starting point and inital heading. More importantly, for each symbol you can also enter a sequence of one or more whitespace-separated Python statements (generally turtle commands) to be executed when that symbol is encountered in the final string.

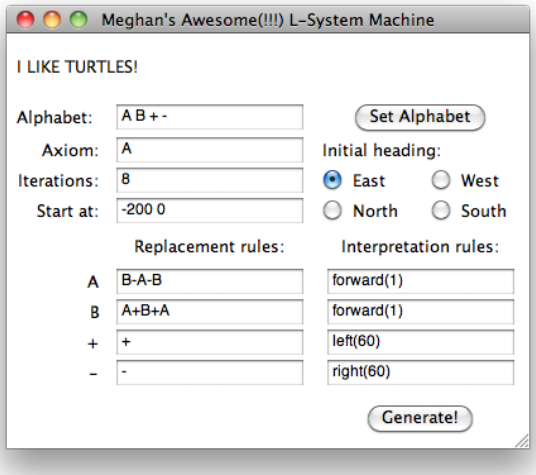

Figure 1: A simple setup.

The most difficult part of implementing the program was verifying the input, not the mechanics of the L-system itself. I tried to make the application as robust as possible, so I could share it with friends.

There are several constraints for valid input. My program deals with all the exceptional cases I could think of, but I'd like to mention one peculiarity. The alphabet may consist of any single-character symbols except the brackets [and] (and whitespace, which is a separator). In my implementation, the two bracket symbols are technically always in the alphabet, but they are considered reserved symbols that correspond to state save and restore operations for use with bracketed L-systems. The user cannot overwrite their rewrite or interpretation rules.

Additional information about input constraints can be found in the readme file associated with the program. The file also contains some scintillating example settings.

1.2 Limitations and Quirks

Warning: This program is incredibly dangerous! Anything you enter as an interpretation rule will be treated like whitespace-separated lines of Python code and blindly executed by the eval() function. This isn't a problem for me, because I trust myself not to execute malicious code on my own laptop. Certainly eval() itself also has limitations which prevents particularly egregious exploits. However, it would be very easy to submit and execute as an interpretation rule something in the spirit of files.deleteAll() or computer.explode(), which makes this program hilariously unsafe.

Visualization is subject to the vagaries of Python's turtle module. If you try to generate a new L-system while the previous L-system is still being drawn, you will get unexpected behavior. The canvas will clear and the turtle will start drawing your new L-system. However, after your new L-system is finished, the turtle will continue to draw a bunch of buffered drawing commands for the previous L-system that were sent out but never drawn on the canvas. To cancel a drawing in progress, close the window as well as any extra windows that pop up afterwards (a related idiosyncrasy). Then you can generate your new L-system on a clean slate.

The program does not support context-sensitive L-systems, parametric L-systems (though wouldn't that be cool), or stochastic L-systems. However, you could stochastically *interpret* an L-system's string by choosing randomly among a set of angles. For example, one of your interpretation rules could be `left(random.choice([30,60,90]))`.

1.3 Results

I gleefully tested my program on many different L-systems. Figure 2 shows some of the bracketed L-systems I generated. Two of them were produced according to grammars given in the textbook (see page 283 of "Bio-Inspired Artificial Intelligence" by Floreano and Mattiussi). One was produced according to a grammar described on the Wikipedia page for L-systems. The last is an interesting pattern I stumbled upon while playing around with the interpretations for one of the plant-like grammars. It doesn't look so plant-like now, does it?

Bracketed L-systems are generally of great interest, given that they can generate lovely naturalistic images. However, I had

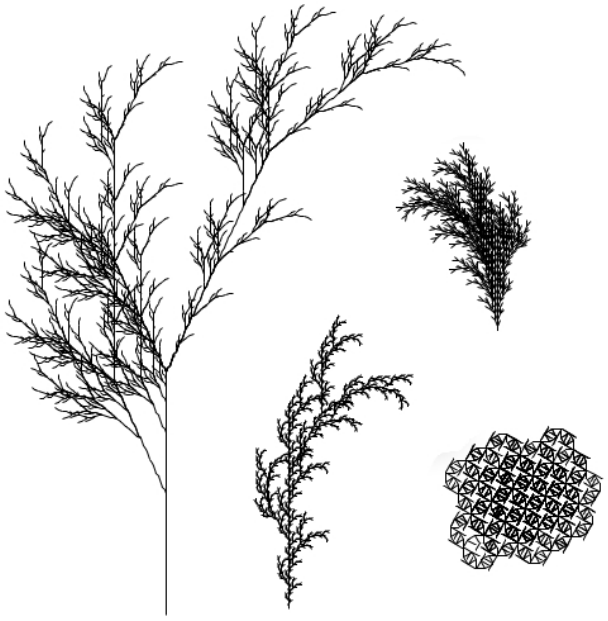

Figure 2: Bracketed L-systems.

more fun experimenting with the interpretations of fractal structures, which in this case did not include bracketing.

Fractals are defined by their regular and self-similar structure. What properties of this structure are preserved between visual interpretations, if any?

I played around with the Koch curve system given in the textbook (see page 279), but the Sierpinski triangle yielded the most interesting results. Formally, the grammar for the Sierpinksi triangle is given by:

$$\begin{aligned} \alpha &= \{A, B, +, -\} \\ \omega &= A \\ \pi &= \{A \to B - A - B, B \to A + B + A, + \to +, - \to -\} \end{aligned}$$

where α is the alphabet, ω is the axiom, and π is the set of rewriting or production rules.

I tried out many different interpretation rules with this grammar. Unsurprisingly, I got a variety of shapes besides the classic triangle. Usually the final shape somehow reflected the underlying symmetry of the string, but not always. The shape on the far right of Figure 3 is asymmetrical, due to an asymmetrical set of interpretation rules. Clearly, interpretation alone has a big impact on the geometric properties of the final shape. A particularly asymmetric interpretation can begin to obscure even the most regular underlying structure!

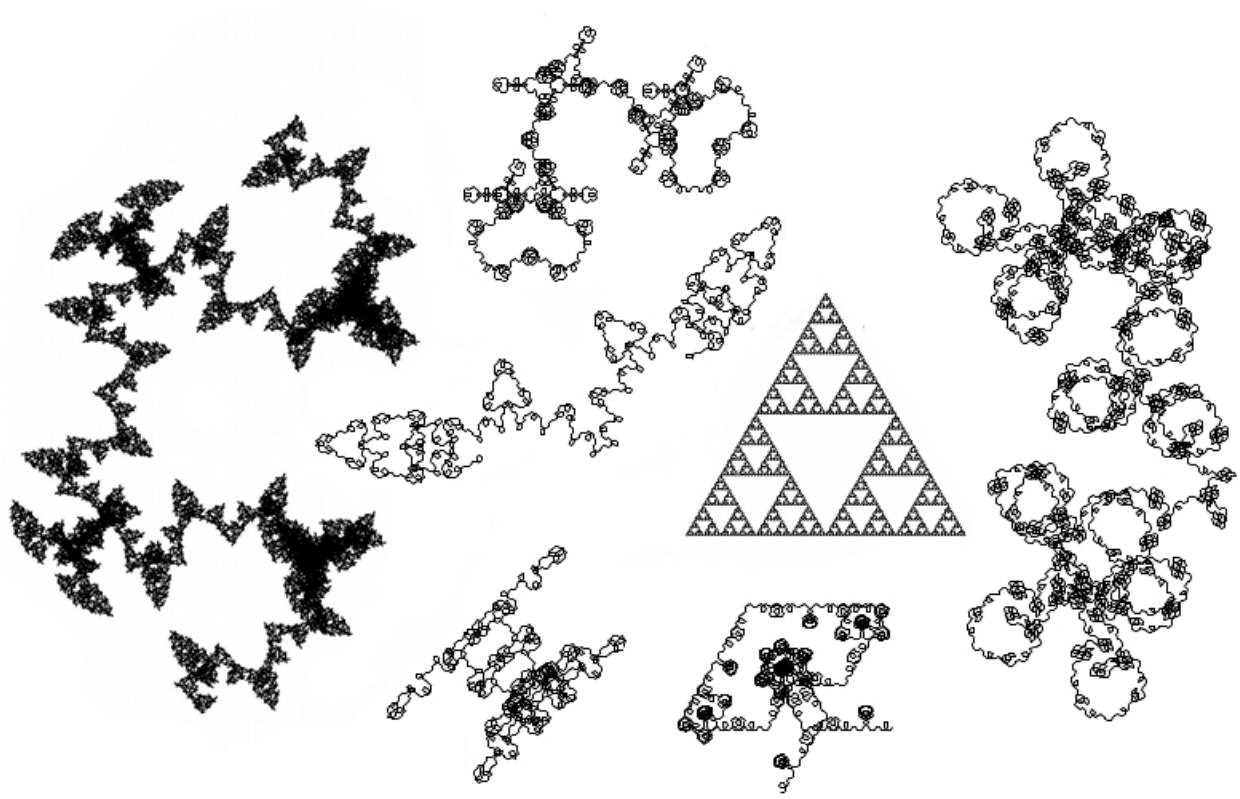

Figure 3: Which one is the Sierpinski triangle? Trick question! They all are!